



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Lab 6: Tuesday 3/14 @ 11:59 pm
 - Homework 8: this Friday @ 11:59 pm
 - Assignment 2: this Friday @ 11:59 pm
 - Support video and slides on Canvas
- Talk by candidate faculty
 - This Wednesday 3/15 @ 10 am at 5317 Sennott Square
 - Donuts will be served!

Previous lecture

- LZW Compression and expansion

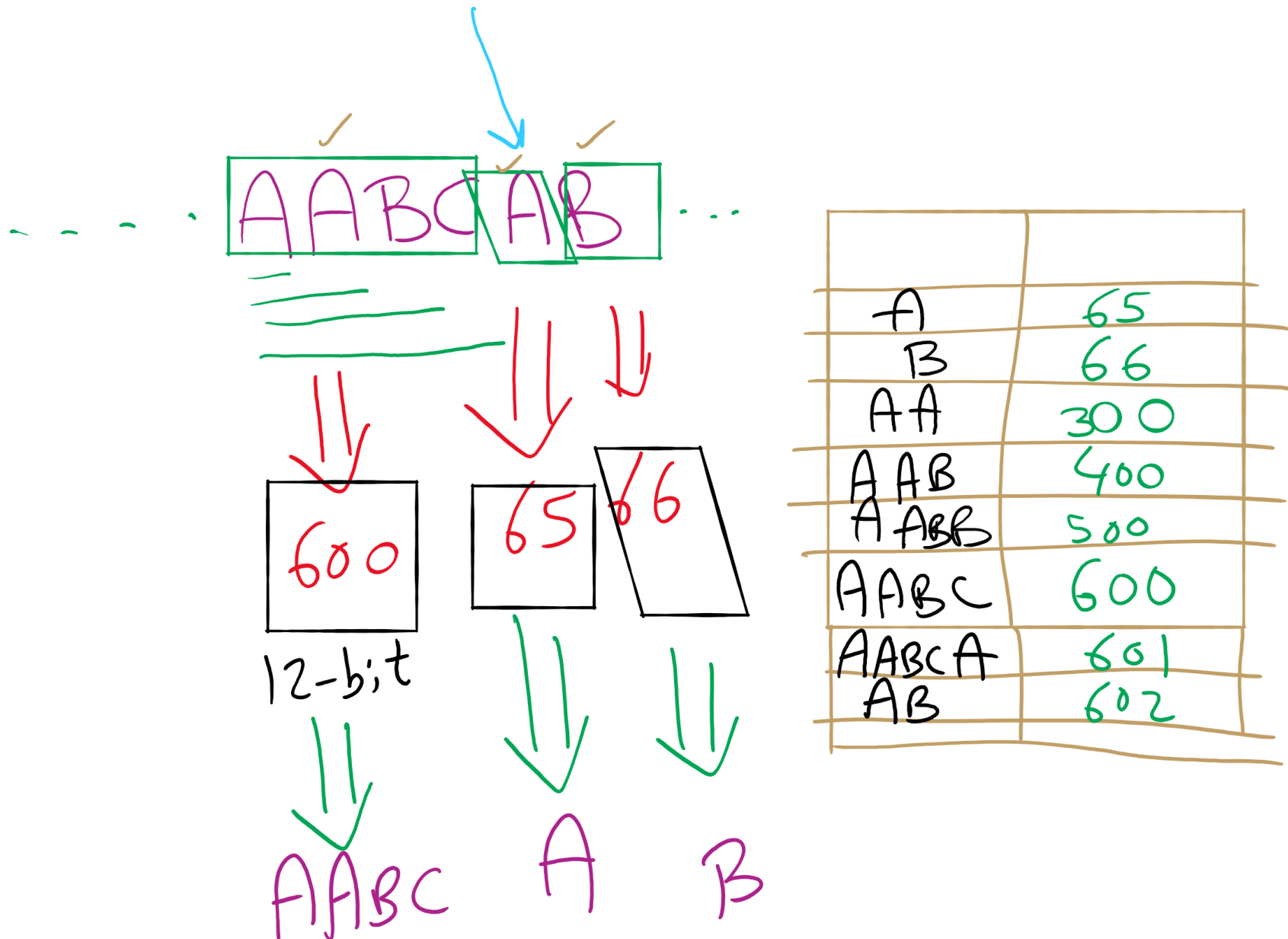
This Lecture

- LZW example and corner case
- Shannon's Entropy
- LZW vs. Huffman
- Burrows-Wheeler Compression Algorithm

LZW Compression

- Both compression and expansion construct the same codebook!
 - Compression stores character string → codeword
 - Expansion stores codeword → character string
 - They contain the same pairs in the same order
 - Hence, the codebook doesn't need to be stored with the compressed file, saving space

LZW Example



Just one tiny little issue to sort out...

- Expansion can sometimes be a step ahead of compression...
 - If, during compression, the (pattern, codeword) that was **just added** to the dictionary is **immediately used** in the next step, the decompression algorithm will not yet know the codeword.
 - This is easily detected and dealt with, however

LZW corner case example

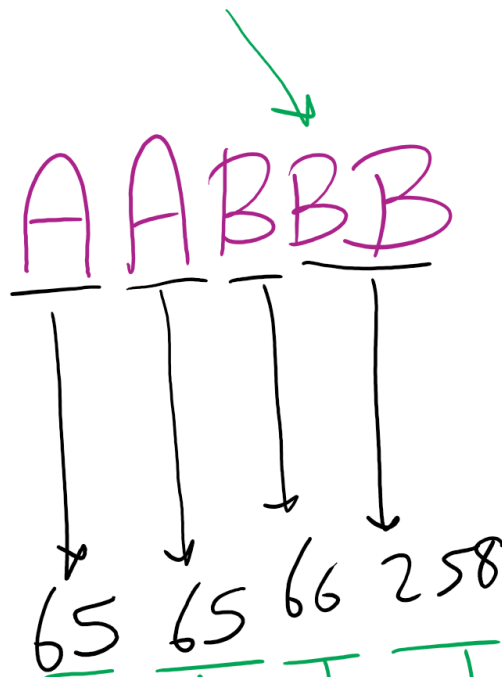
- Compress, using 12 bit codewords: AAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	256:AA
257	AAA	257:AAA

LZW Corner Case



A	65
✓ AA	256
AB	257
BB	258

A A B BB

65	A
66	B
✓ 256	AA
257	AB
258	BB

↑ Codeword ↓
2

Prev output + 1st character of
Prev output

LZW implementation concerns: codebook

- How to represent/store during:
 - Compression
 - Expansion
- Considerations:
 - What operations are needed?
 - How many of these operations are going to be performed?
- Discuss

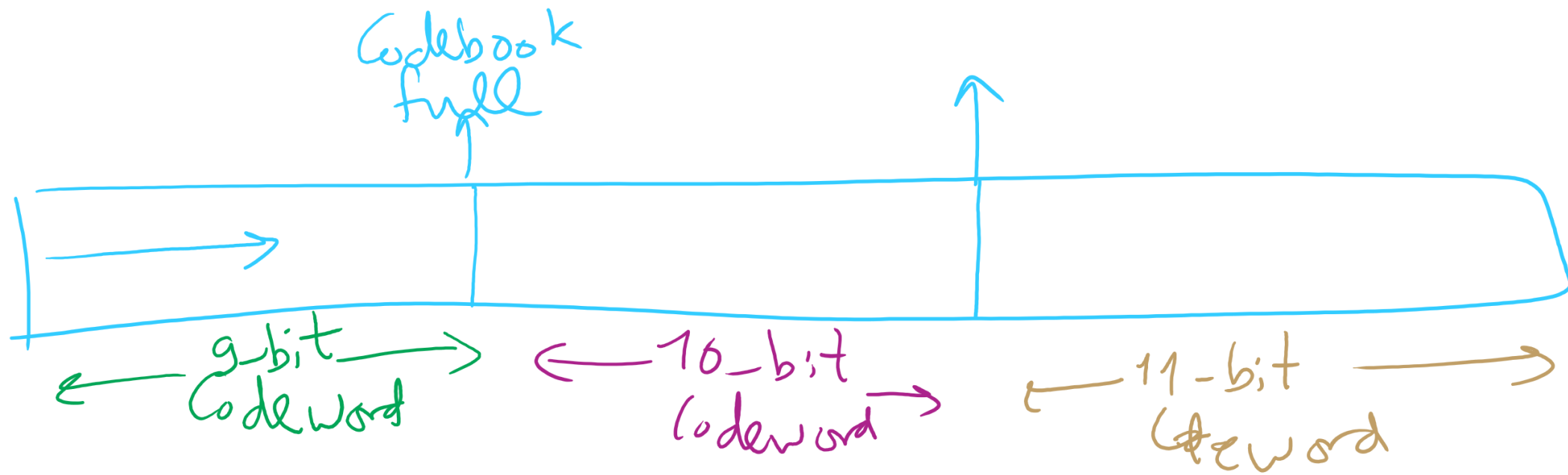
Further implementation issues: codeword size

- How long should codewords be?
 - Use fewer bits:
 - Gives better compression earlier on
 - But, leaves fewer codewords available, which will hamper compression later on
 - Use more bits:
 - Delays actual compression until longer patterns are found due to large codeword size
 - More codewords available means that greater compression gains can be made later on in the process

Variable width codewords

- This sounds eerily like variable length codewords...
 - Exactly what we set out to avoid!
- Here, we're talking about a different technique
- Example:
 - Start out using 9 bit codewords
 - When codeword 512 is inserted into the codebook, switch to outputting/grabbing 10 bit codewords
 - When codeword 1024 is inserted into the codebook, switch to outputting/grabbing 11 bit codewords...
 - Etc.

Adaptive Codeword Size

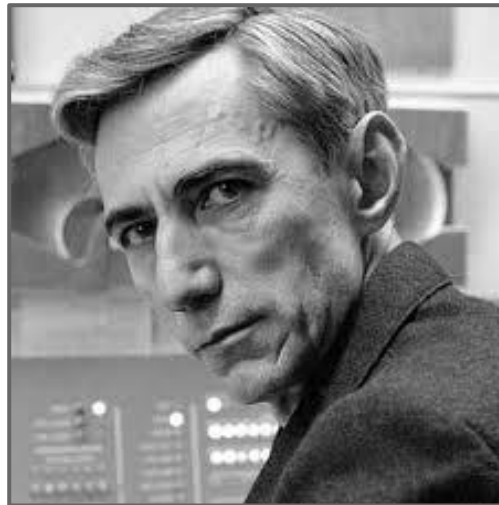


Even further implementation issues: codebook size

- What happens when we run out of codewords?
 - Only 2^n possible codewords for n bit codes
 - Even using variable width codewords, they can't grow arbitrarily large...
- Two primary options:
 - Stop adding new keywords, use the codebook as it stands
 - Maintains long already established patterns
 - But if the file changes, it will not be compressed as effectively
 - Throw out the codebook and start over from single characters
 - Allows new patterns to be compressed
 - Until new patterns are built up, though, compression will be minimal

Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy



Information theory in a single slide...

- Founded by Claude Shannon in his paper "A Mathematical Theory of Communication"
- **Shannon Information** is a measure of the **unpredictability of information content**
 - Example: which is more unpredictable?
 - a character that occurs with probability 0.5 or
 - a character that occurs with probability 0.25
 - which should have more entropy?

Shannon Information

- Shannon Information of a message m
 - $I(m) = -1 * \log_2 \text{Pr}(m)$ bits
 - $\text{Pr}(m)$ is the probability of message m
- Examples:
 - $\text{Pr}(c1) = 0.5 \rightarrow I(c1) = -1 * \log_2(0.5) = -1 * -1 = 1$ bit
 - $\text{Pr}(c2) = 0.25 \rightarrow I(c2) = -1 * \log_2(0.25) = -1 * -2 = 2$ bits
 - $\text{Pr}(c3) = 1/2^{100} \rightarrow I(c3) = -1 * \log_2(2^{-100}) = -1 * -100 = 100$ bits

Shannon's Information Entropy

- **Shannon's Entropy** is a key measure in information theory
 - Slightly different from thermodynamic entropy
- Entropy of an information source (e.g., a file)
 - $H = \sum_{\text{all unique messages } m} Pr(m) * I(m)$
 - average of Shannon's information of all unique messages
- **Entropy per bit** = $H / \text{file size in bits}$

Entropy of a file

How can we determine the probability of each character in the file?

- if it depends only on file contents

- $Pr(c) = f(c) / \text{file size}$

- However, may also depend on receiver and sender contexts and their world knowledge

Implications on Lossless Compression

- By losslessly compressing data, we represent the same information in less space
 - entropy of original file = entropy of compressed file
- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of entropy per bit of compressed message

Entropy applied to language

- **Entropy of a language:** the average number of bits required to store a letter of the language
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per letter
- Entropy of a language * length of message in characters = amount of information contained in that message

The showdown you've all been waiting for...

HUFFMAN vs LZW

- In general, LZW will give better compression
 - Also better for compressing archived directories of files
 - Why?
 - Very long patterns can be built up, leading to better compression
 - Different files don't "hurt" each other as they did in Huffman
 - Remember our thoughts on using static tries?

So lossless compression apps use LZW?

- Well, gifs can use it
 - And pdfs
- Most dedicated compression applications use other algorithms:
 - DEFLATE (combination of LZ77 and Huffman)
 - Used by PKZIP and gzip
 - Burrows-Wheeler transforms
 - Used by bzip2
 - LZMA
 - Used by 7-zip
 - brotli
 - Introduced by Google in Sept. 2015
 - Based around a " ... combination of a modern variant of the LZ77 algorithm, Huffman coding[,] and 2nd order context modeling ... "

Is there a universal compression algorithm?

- That can compress every file and any file?
- Nope!
- No algorithm can compress every bitstream
 - Assume we have such an algorithm
 - We can use to compress its own output!
 - And we could keep compressing its output until our compressed file is 0 bits!
 - Clearly this can't work
- Proofs in Proposition 5 of Section 5.5 of the text

What about the best compression algorithm for a given file?

- Nope!
- This problem is undecidable
- Example:
 - A Fibonacci sequence of one billion numbers can be compressed by a program to generate Fibonacci numbers

A final note on compression evaluation

- "Weissman scores" are a made-up metric for Silicon Valley (TV)



Burrows-Wheeler Data Compression Algorithm

- **Best** compression algorithm (in terms of compression ratio) **for text**
- The basis for UNIX's **bzip2** tool

Adapted from: <https://www.cs.princeton.edu/courses/archive/spr03/cos226/assignments/burrows.html>

BWT: Compression Algorithm

- Three steps
 - Burrows-Wheeler Transform
 - Cluster same letters as close to each other as possible
 - Move-To-Front Encoding
 - Convert output of previous step into an integer file with **large frequency** differences
 - Huffman Compression
 - Compress the file of integers using Huffman Compression

BWT: Expansion Algorithm

- Apply the inverse of compression steps in reverse order
 - Huffman decoding
 - Move-To-Front decoding
 - Inverse Burrows-Wheeler Transform

Move-To-Front Encoding

- Initialize an ordered list of the 256 ASCII characters
 - character i appears i th in the list
- For each character c from input
 - output the index in the list where c appears
 - move c to the front of the list (i.e., index 0)

Move-To-Front Encoding

Input:

e a e d e e

0

a

1

b

2

c

3

d

4

e

Output:

Move-To-Front Encoding

Input:

e a e d e e

0

a

1

b

2

c

3

d

4

e

Output:

4

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

1

b

a

2

c

b

3

d

c

4

e

d

Output:

4

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

a

1

b

a

e

2

c

b

b

3

d

c

c

4

e

d

d

Output:

4

1

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

a

e

1

b

a

e

a

2

c

b

b

b

3

d

c

c

c

4

e

d

d

d

Output:

4

1

1

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

a

e

d

1

b

a

e

a

e

2

c

b

b

b

a

3

d

c

c

c

b

4

e

d

d

d

c

Output:

4

1

1

4

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

a

e

d

e

1

b

a

e

a

e

d

2

c

b

b

b

a

a

3

d

c

c

c

b

b

4

e

d

d

d

c

c

Output:

4

1

1

4

1

Move-To-Front Encoding

Input:

e a e d e e

0

a

e

a

e

d

e

e

1

b

a

e

a

e

d

d

2

c

b

b

b

a

a

a

3

d

c

c

c

b

b

b

4

e

d

d

d

c

c

c

Output:

4

1

1

4

1

0

Move-To-Front Encoding

In the output of MTF Encoding, smaller integers have higher frequencies than larger integers

Move-To-Front Decoding

- Initialize an ordered list of 256 characters
 - same as encoding
- For each integer i (i is between 0 and 255)
 - print the i th character in the list
 - move that character to the front of the list

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0

a

1

b

2

c

3

d

4

e

Output:

e

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0

a

e

1

b

a

2

c

b

3

d

c

4

e

d

Output:

e

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

e a

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

e a e

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0

a

e

a

e

d

1

b

a

e

a

e

2

c

b

b

b

a

3

d

c

c

c

b

4

e

d

d

d

c

Output:

e

a

e

d

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

e a e d e

Move-To-Front Decoding

- Decoding

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

e a e d e e

BWT: Compression Algorithm

- **Compression**

- Burrows-Wheeler Transform
- Move-To-Front Encoding ✓
- Huffman Compression ✓

- **Expansion**

- Huffman decoding ✓
- Move-To-Front decoding ✓
- Inverse Burrows-Wheeler Transform

Burrows-Wheeler Transform

- **Rearranges** the characters in the input
 - lots of clusters with **repeated characters**
 - still possible to **recover** the original input
- Intuition: Consider the string **hen** in English text
 - most of the time the letter preceding it is t or w
 - group all such preceding letters together (mostly t's and some w's)

Burrows-Wheeler Transform

- For each block of length N characters
 - generate **N strings** by **cycling** the characters of the block one step at a time
 - **sort** the strings
 - output is the **last column** in the sorted table and the **index** of the original block in the sorted array

Burrows-Wheeler Transform

- Example: Let's transform "ABRACADABRA"

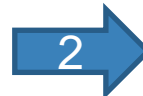
- $N = 11$

- Cyclic Versions of the string:

- ABRACADABRA
- BRACADABRAA
- RACADABRAAB
- ACADABRAABR
- CADABRAABRA
- ADABRAABRAC
- DABRAABRACA
- ABRAABRACAD
- BRAABRACADA
- RAABRACADAB
- AABRACADABR

- After Sorting

- AABRACADABR
- ABRAABRACAD
- ABRACADABRA
- ACADABRAABR
- ADABRAABRAC
- BRAABRACADA
- CADABRAABRA
- DABRAABRACA
- RAABRACADAB
- RACADABRAAB



RDARCAAAABB

Burrows-Wheeler Transform Example 2

- Input: ABABABA
- **Step 1: Build an array of 7 strings, each a circular rotation of the original by one character**
 - ABABABA
 - BABABAA
 - ABABAAB
 - BABAABA
 - ABAABAB
 - BAABABA
 - AABABAB
- **Step 2: Sort the array alphabetically**
 - **Notice that** the first column of the sorted array has the same characters as the last column
 - all columns have the same set of letters
- **Step 3: Output the last column of the sorted array and the index of the input string in the sorted array**

original array

ABABABA

BABABAA

ABABAAB

BABAABA

ABAABAB

BAABABA

AABABAB

sorted array

AABABAB

ABAABAB

ABABAAB

ABABABA

BAABABA

BABAABA

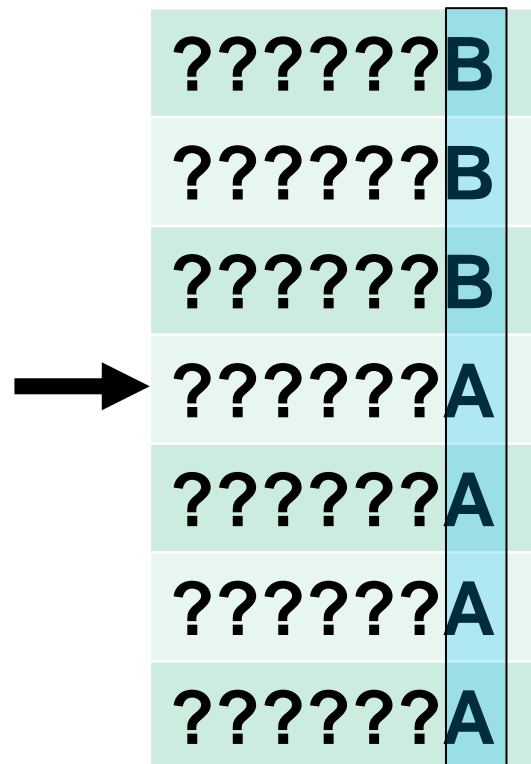
AABABAB



Output of BWT:
BBBAAAA and 3

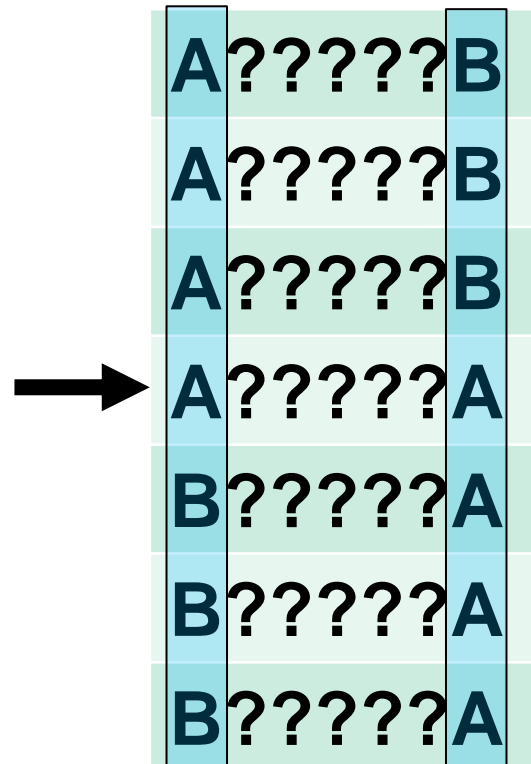
Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
 - BBBAAAA → AAAABBB
 - The first column of the sorted array has the same characters as the last column
 - but in sorted order



Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
 - BBBAAAA → AAAABBB
 - This gives us the first column of the sorted array



Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - holds the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

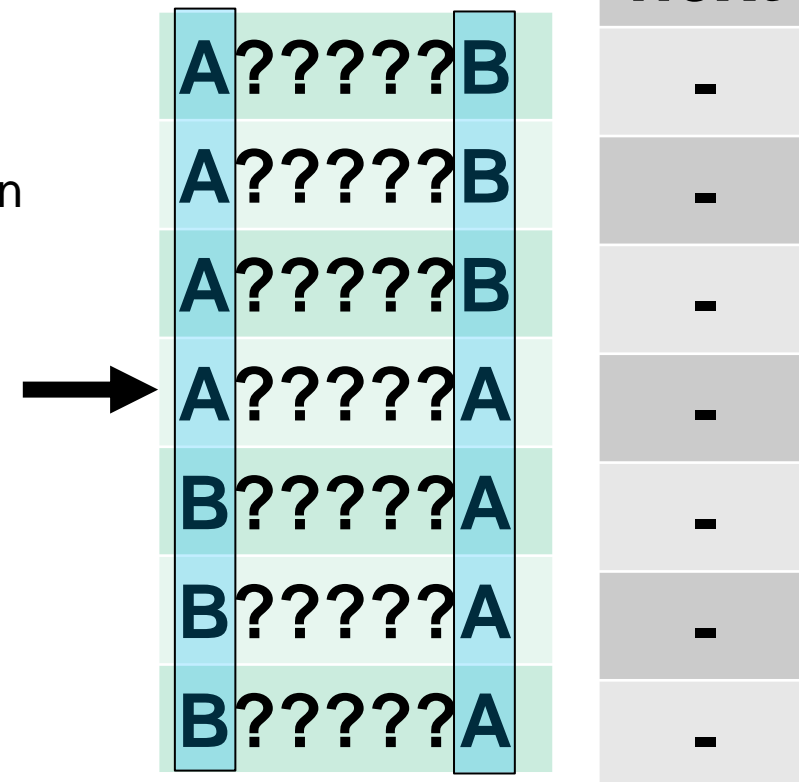
ABABABA
BABABAA
ABABAAB
BABAABA
ABAABAB
BAABABA
AABABAB



A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$



	next
A?????B	-
A?????B	-
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
-
-
-
-
-
-

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
4
-
-
-
-
-

Burrows-Wheeler Transform Decoding

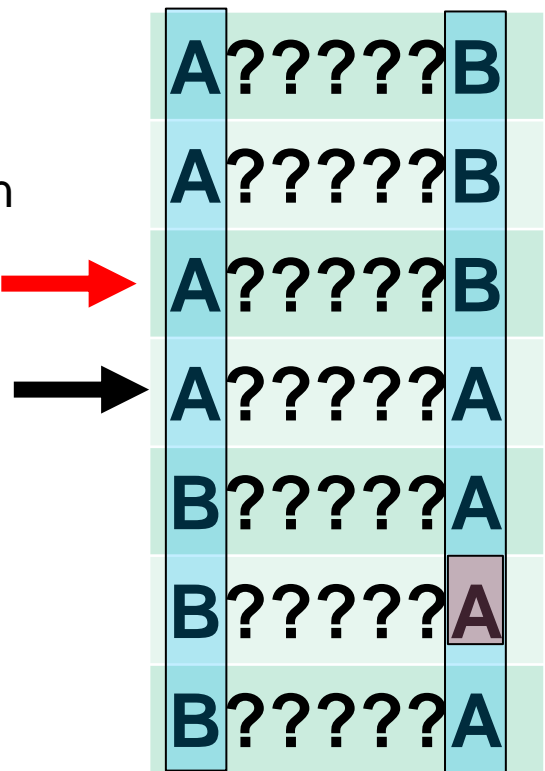
- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
4
-
-
-
-
-

Burrows-Wheeler Transform Decoding

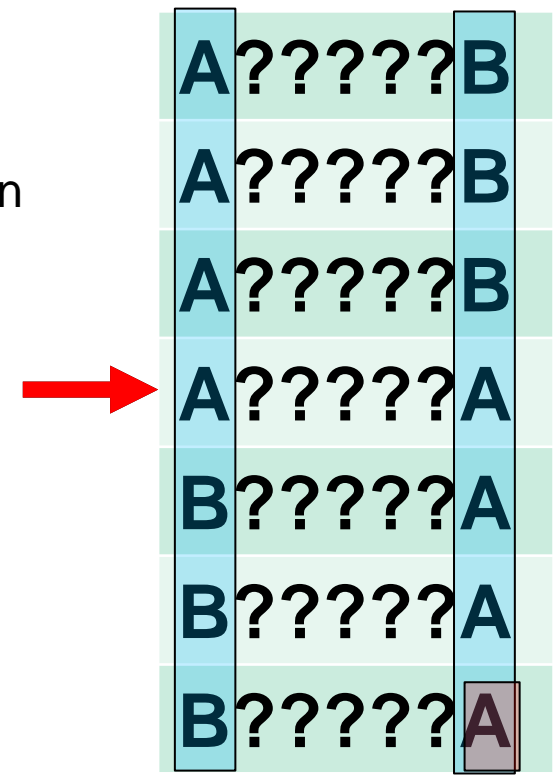
- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$



next
3
4
5
-
-
-
-

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

	A	?????	B		next
→	A	?????	B		3
	A	?????	B		4
	A	?????	B		5
→	A	?????	A		6
→	B	?????	A		0
	B	?????	A		-
	B	?????	A		-

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
4
5
6
0
1
-

Burrows-Wheeler Transform Decoding

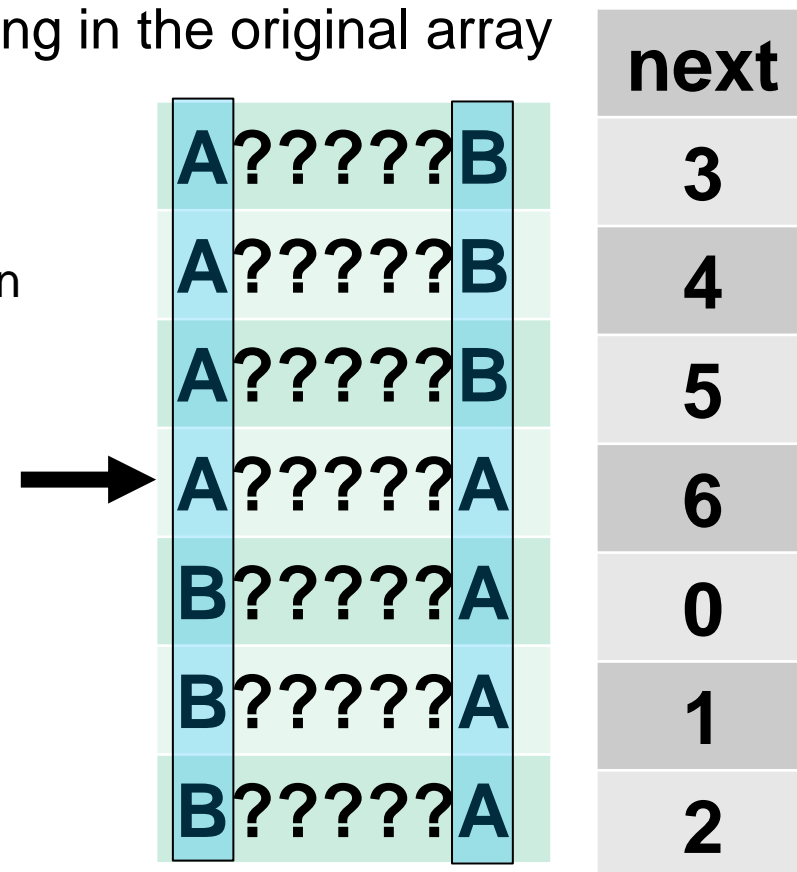
- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$

A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
 - defined for each entry in the sorted array
 - tells us the index in sorted array of the next string in the original array
 - Scan through the first column
 - for each row i holding character c
 - $\text{next}[i] = \text{first unassigned index of } c \text{ in the last column}$
- Why does that work?
 - first character of a string becomes the last character in the next string in the original order



The diagram illustrates the process of filling the next array. It shows a sorted array of strings with the first column highlighted in blue and the last column highlighted in green. An arrow points from the first column to the next array, which is a vertical list of indices. The next array is filled with the index of the first occurrence of the character from the first column in the last column.

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?

A??????



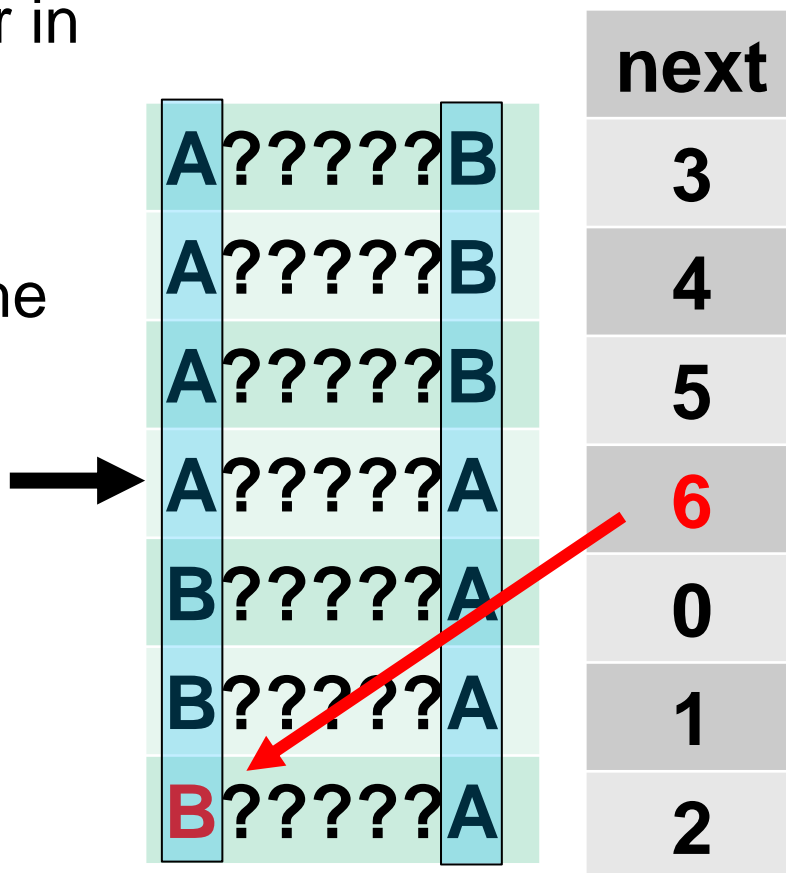
A	?????	B
A	?????	B
A	?????	B
A	?????	A
B	?????	A
B	?????	A
B	?????	A

next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?
- The next character is the first character of the next string in the original order
 - first character in string at next[3]

AB?????

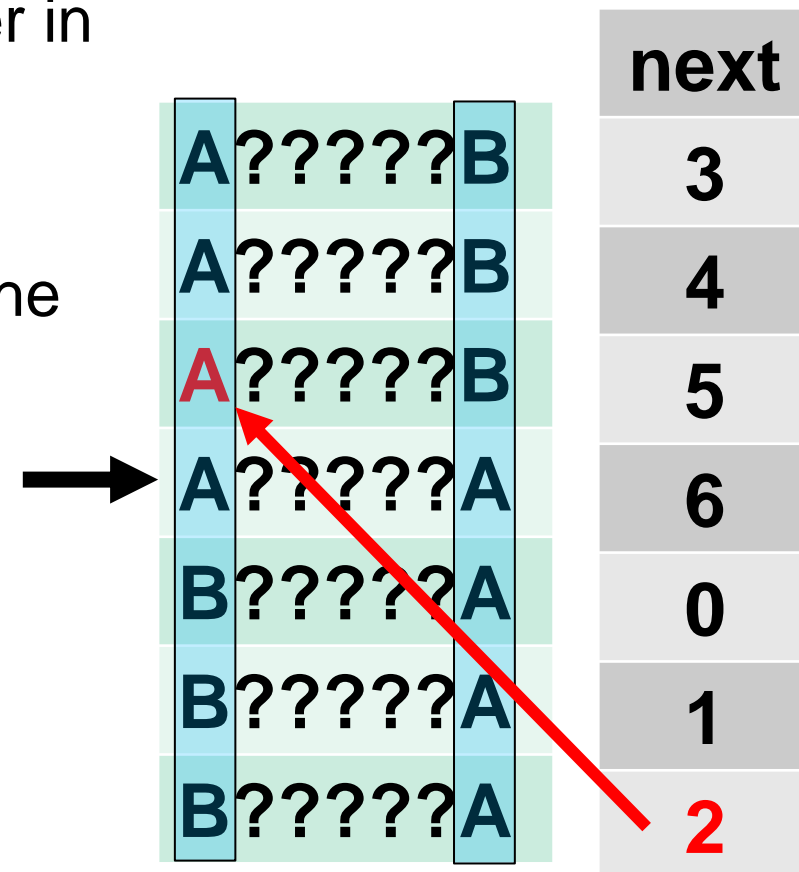


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?
- The next character is the first character of the next string in the original order
 - first character in string at next[6]

ABA????

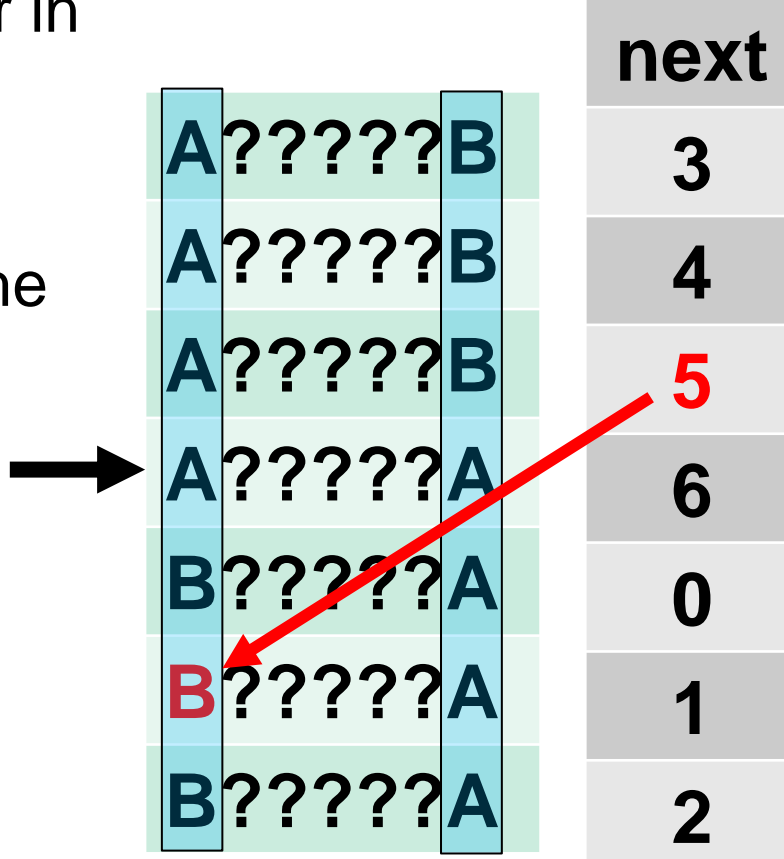


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?
- The next character is the first character of the next string in the original order
 - first character in string at next[2]

ABAB???

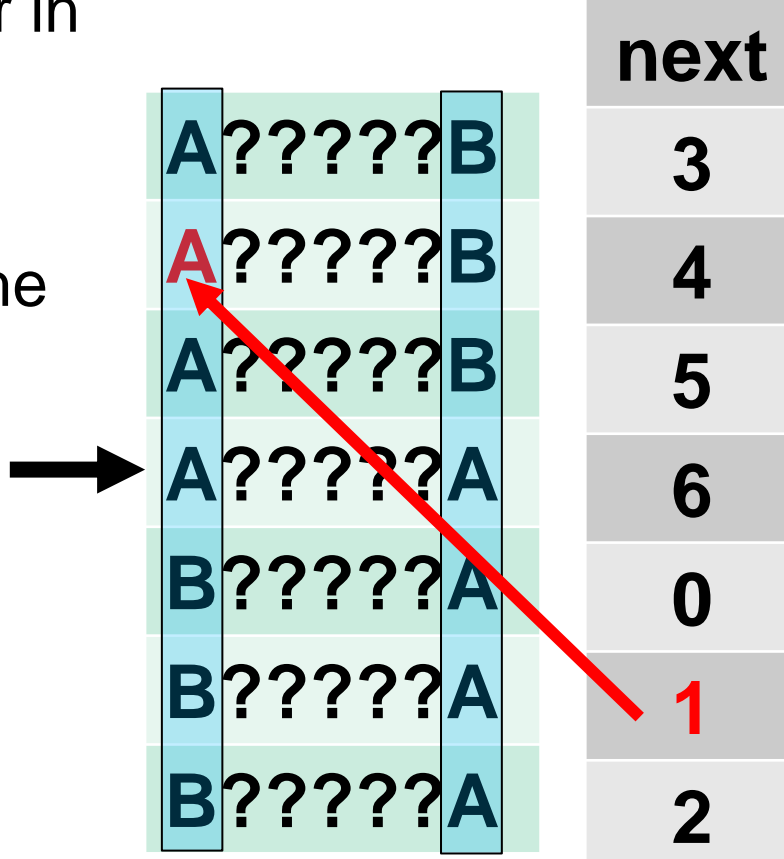


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?
- The next character is the first character of the next string in the original order
 - first character in string at next[5]

ABABA??



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
 - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
 - why?
- The next character is the first character of the next string in the original order
 - first character in string at next[5]

ABABABA



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Downsides of Burrows-Wheeler Algorithm

- process **blocks** of input file
 - Compared to LZW, which processes the input **one character at time**
- The **larger** the block size, the **better** the compression
 - But the **longer** the sorting time